Sistemi Distribuiti e Cloud Computing
Progetto A2: Applicazione basata su microservizi
Secure PassWorld

Enrico D'Alessandro
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
enrico.dalessandro@alumni.uniroma2.eu

Alessandro De Angelis
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
alessandro.deangelis.97@alumni.uniroma2.eu

Pierpaolo Spaziani
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
pierpaolo.spaziani@alumni.uniroma2.eu

I. Introduzione

Il progetto realizzato consiste in un'applicazione con architettura a microservizi, che fornisce una piattaforma web per la gestione di password sicure e dei tool di supporto alla sicurezza per le aziende.

In particolare, gli utenti attraverso la piattaforma web possono:

- Generare nuove password.
- Gestire le password salvate.
- Richiedere e gestire richieste di password condivise (*solo per utenti appartenenti a gruppi*).

I servizi offerti alle aziende permettono di:

- Utilizzare il tool di doppia autenticazione.
- Utilizzare le password generate in modo condiviso, per limitare l'acceso ad aree riservate.

II. Architettura

L'applicazione è composta da 7 microservizi di cui uno di tipo stateful. È stata sviluppata in Python e adotta per la comunicazione sincrona, il framework di chiamata a procedura remota *gRPC*, invece per la comunicazione asincrona, il middleware message-oriented *RabbitMQ*. Quest'ultimo permette di avere disaccoppiamento spaziale e temporale tra i microservizi. Ogni microservizio è stato instanziato all'interno di un container. È stato utilizzato *Docker* per la creazione delle immagini dei container e *Kubernetes* come tool di orchestrazione. Quest'ultima scelta è stata dettata dalla flessibilità che il sistema offre, data la necessità di dover rendere l'applicazione scalabile in base al numero di richieste e tollerante ai guasti di tipo crash. Per lo sviluppo, sono stati inoltre rispettati 2 design pattern: *Circuit-breaker* e *Database-per-service*.

L'architettura di tale applicazione si basa sulla divisione delle funzionalità in due scenari differenti. Ci si riferirà, da questo punto in poi, ad account di tipo '*Classic*' ed account di tipo '*Enterprise*' per gli utenti, e ad account '*Agency*' per le società private.

Gli account *Classic* possono utilizzare la piattaforma come un semplice password manager che permette la creazione, memorizzazione e gestione delle password. Gli account *Enterprise*, sono strettamente legati ad account di tipo *Agency*. In particolare, un utente che viene registrato da un *Agency* come suo 'dipendente' passa da un account di tipo *Classic* ad uno di tipo *Enterprise*. Quest'ultimo permette di accedere a delle funzionalità avanzate, ovvero la possibilità di richiedere password condivise ad un gruppo.

Lo scopo e la funzionalità di tale suddivisione, sta nel fatto di mettere a disposizione delle società private, dei servizi di sicurezza senza doverli implementare loro stessi. In particolare, viene offerto un meccanismo di doppia autenticazione e la possibilità di accedere ad aree riservate con una password condivisa temporanea richiesta sulla nostra piattaforma.

III. Microservizi

A. Login

Il microservizio *Login* permette di effettuare la registrazione ed il conseguente login, sia degli utenti che delle aziende, per l'accesso alla piattaforma web. Gestisce inoltre il collegamento tra le *Agency* e i propri 'dipendenti', ovvero utenti con account *Enterprise*.

B. New Password

Il microservizio *New Password* permette agli utenti di generare password sicure scegliendo tra differenti configurazioni. In particolare, oltre ad indicare la lunghezza, è possibile scegliere di creare password contenenti caratteri alfanumerici, solo caratteri alfabetici oppure solo numeri. Grazie alla comunicazione sincrona con il microservizio *Password Manager*, prima della generazione della password, l'utente può decidere se salvarla e a quale servizio associarla.

C. Password Manager

Il salvataggio delle password è reso disponibile anche per password non generate dalla piattaforma. L'utente può memorizzare direttamente una propria password specificandone il servizio a cui essa è associata. Questo lascia agli utenti la libertà di utilizzare la piattaforma web senza limitazioni. Il salvataggio di una password associata ad un servizio per cui ne è già presente una, comporta la modifica di quest'ultima. Grazie a tale microservizio l'utente ha ovviamente anche la possibilità di visualizzare tutte le password memorizzate a suo nome.

D. Group Manager

Per quanto riguarda la creazione di gruppi di utenti, questa funzionalità viene offerta dal microservizio *Group Manager*. Un'*Agency*, ha la possibilità di registrare utenti tra i propri dipendenti. Quest'operazione comporta il passaggio degli account di tali utenti da uno di tipo *Classic* ad uno di tipo *Enterprise*. Inoltre, un'*Agency*, ha la possibilità di creare gruppi di utenti ed associare loro dei nomi. I gruppi creati possono essere formati soltanto da utenti che risultano essere dipendenti presso quella determinata azienda.

E. Shared Password

Tramite il microservizio *Shared Password* un utente *Enterprise* può richiedere una password temporanea utilizzabile sul sito di terze parti, ad un gruppo a cui appartiene. In seguito alla richiesta dell'utente, le successive comunicazioni tra i microservizi avvengono sia in maniera asincrona che sincrona. In particolare, tramite *RabbitMQ*, i messaggi asincroni vengono scambiati attraverso 3 code differenti: *request_queue* (da *Shared Password* a *Group Manager*), *info_queue* (da *Group Manager* a *Shared Password*) ed *email_queue* (da *Shared Password* a *Notification*). Invece, tra *Shared Password* e *New Password*, la comunicazione avviene tramite *gRPC*. Essendo un microservizio stateful, il suo comportamento varia in base allo stato della richiesta. Nel caso in cui non esista alcuna password valida da restituire all'utente, *Shared Password* pubblica un messaggio di richiesta per il microservizio *Group Manager* (*request_queue*). Quest'ultimo si occupa di reperire gli indirizzi email degli altri partecipanti al gruppo e di inoltrarli a *Shared Password* (*info_queue*). Il microservizio *Shared Password*, successivamente consuma dalla coda il messaggio contenente l'elenco degli altri membri. A questo punto, richiede la creazione di un token (6 caratteri alfanumerici) per ogni membro, tramite il microservizio *New Password*. Fatto ciò, pubblica per *Notification* (*email_queue*) le informazioni da inviare via email ad ogni componente del gruppo. In questo modo, ogni membro riceve un'email contenente il token da inserire nell'apposita interfaccia per accettare o declinare la richiesta. La password viene generata solo se tutti i partecipanti del gruppo accettano, anche un solo rifiuto comporta l'annullamento della richiesta. In entrambi i casi, viene inviata un'email al richiedente indicante se la password è stata accettata, oppure se è stata rifiutata. Un secondo comportamento del microservizio, è quando la richiesta è stata già inviata e si trova in uno stato pendente. In questo caso comunica all'utente il numero dei membri che hanno già accettato. L'ultimo caso, è quando tutti i membri del gruppo hanno accettato e viene restituita la password (8 caratteri alfanumerici con simboli) utilizzabile sul sito di terze parti.

F. Double Authentication

Il microservizio *Double Authentication* fornisce ai siti di terze parti abilitati sulla nostra piattaforma, un servizio di doppia autenticazione. *Double Authentication* riceve come informazioni l'email dell'utente che sta effettuando il login sul sito di terze parti e il nome del sito stesso. Successivamente provvede all'invio di un token via email che l'utente potrà inserire per completare la procedura di autenticazione. In particolare, per la generazione del token (5 numeri), il microservizio *Double Authentication* comunica tramite *gRPC* con il microservizio *New Password*. Dopodiché, pubblica su *RabbitMQ* (*email_queue*) le informazioni utili a *Notification* per l'invio del codice via email. Il microservizio si occupa infine anche del controllo della validità del codice inserito dall'utente sul sito di terze parti.

G. Notification

Il microservizio *Notification*, ricevendo richieste esclusivamente tramite *RabbitMQ* (*email_queue*), si occupa dell'invio delle email descritte nei microservizi precedenti.

IV. API Gateway

A. WebApp

Offre le *API* per quei microservizi direttamente accessibili dal client attraverso l'interfaccia web, la quale viene gestita da un server web *Flask*.

Le *API* accessibili sono:

- */home*
- */login*
- */register*
- */homepage*
- */newpassword*
- */savepassword*
- */listpassword*
- */grouplist*
- */groupcreate*
- */addEmployee*
- */notification*
- */logout*

B. PublicAPI

Offre le *API* per i microservizi utilizzabili da applicazioni terze. Tramite questo gateway, i privati possono sfruttare i servizi offerti a loro: *Doppia Autenticazione* e *controllo Password Condivisa*.

Le *API* accessibili sono:

- */shared_pass*
- */double_auth*
- */register_user*

V. Design Pattern

A. Circuit-breaker

Per prevenire che un problema/malfunzionamento su un determinato microservizio si ripercuota in cascata sugli altri che vanno ad invocarlo, è stato adottato il patter *Circuit-breaker*. Secondo questo pattern, l'invocazione ad un microservizio non è diretta, ma mediata da un proxy introdotto. Il suo ruolo è quello di aprire il circuito nel momento in cui il servizio invocato subisce un malfunzionamento. Il *Circuit-breaker*, nel momento in cui il numero di tentativi falliti di contattare il supplier supera un determinata soglia (nel progetto impostata a 2), apre il circuito interrompendo la comunicazione tra il client ed il servizio per un determinato timeout (nel progetto impostato a 5 secondi). Dopo la scadenza del timeout, l'interruttore consente il passaggio di un numero limitato di richieste di test. Se tali richieste hanno esito positivo, l'interruttore riprende il normale funzionamento. Altrimenti, se c'è un fallimento, il periodo di timeout riparte. In questo modo il proxy è in grado di restituire rapidamente un messaggio di servizio non disponibile senza che il client vada in timeout per ogni tentativo di richiesta non andata a buon fine. Nell'applicazione tale pattern è stato applicato in ogni chiamata *gRPC* tra i microservizi.

B. Database-per-service

Per gestire lo stato legato ai diversi microservizi, è stato adottato il pattern *Database-per-service*. Tale pattern prevede l'utilizzo di un database privato per ogni microservizio che lo necessiti. Ciascun database è accessibile soltanto tramite la sua API da parte del microservizio che lo utilizza.

VI. Piattaforma software

A. Docker

Per lo sviluppo di tale applicazione è stato utilizzato *Docker*, ovvero una piattaforma software che consente la

virtualizzazione a livello di sistema operativo e che permette di creare, testare e distribuire applicazioni con rapidità. L'utilizzo che ne è stato fatto è quello della creazione di immagini per i container da instanziare poi all'interno del cluster di *Kubernetes*.

B. Kubernetes

Kubernetes è una piattaforma per l'orchestrazione e gestione di container. La scelta di utilizzare tale piattaforma è dovuta alla capacità di *Kubernetes* di essere self-healing (*auto-placement*, *auto-restart*, *auto-replication*, *auto-scaling* dei container). Più in particolare *Kubernetes* permette la schedulazione di unità minime dette *Pod* all'interno dei quali possono risiedere uno o più container. Nel caso di questa applicazione in ogni pod risiede un solo container.

I componenti utilizzati all'interno di *Kubernetes* per lo sviluppo e gestione di tale applicazione sono:

- *Deployment*
- *Service*
- *Statefulset*
- *Persistent Volumes*

Un *Deployment* è un componente di *Kubernetes* che consente di descrivere il ciclo di vita dell'applicazione specificando, ad esempio, le immagini da utilizzare, il numero di pod necessari e relative modalità di aggiornamento.

Kubernetes offre una virtual network, ovvero ad ogni *Pod* è associato un indirizzo IP. Ciascun *Pod* può comunicare con gli altri proprio grazie a questo indirizzo IP. Quando un Pod muore e viene rimpiazzato con un nuovo *Pod*, l'indirizzo IP associato a quel *Pod* verrà ricreato. Questo può essere uno svantaggio se per esempio su di un *Pod* avevamo instanziato il container per il nostro database. Per questo motivo K8s ha un altro componente per far fronte a questo problema. Il componente prende il nome di *Service*.

Il *Service* permette di disaccoppiare il ciclo di vita di un *Pod* e il *Service* stesso. Questo significa che, anche se un *Pod* muore, il *Service* e l'indirizzo IP associati al *Pod* resteranno gli stessi. Definisce quindi come esporre dei *Pod* su una rete interna o esterna.

Gli *Statefulset* sono utilizzati per gestire applicazioni stateful, in particolare si occupano della gestione del *Deployment* e dello scaling di un insieme di pod, fornendo una garanzia di ordinamento e unicità di tali pod. A differenza del *Deployment* uno *Statefulset* mantiene un'identificatore numerico incrementale.

I *Persistent Volumes* sono una risorsa di *Kubernetes* che permettono di aggiungere uno strato di persistenza al cluster di kubernetes, poiché permettono di mantenere i dati indipendentemente al ciclo di vita dei nodi facenti parte del cluster. Questo poiché i dati di tali *Persistent Volumes* sono esterni al cluster e residenti, nel nostro specifico caso, sulla macchina host. Per l'utilizzo di *Persistent Volumes* è necessario un *Persistent Volume Claim*, ovvero una richiesta di storage da parte di un utente. La classe di storage utilizzata all'interno dell'applicazione sviluppata è quella locale.

L'autoscaling di *Kubernetes* utilizzato in questa applicazione è l'*Horizontal Pod Autoscaler* il quale permette di scalare il numero di repliche dei *Pod*. L'algoritmo utilizzato si basa sull'osservazione dell'utilizzazione della CPU che, insieme all'utilizzo di memoria, costituiscono le uniche metriche nativamente supportate da *Kubernetes*. Il numero minimo di repliche impostato per ogni microservizio è di una, mentre il numero massimo è di cinque. L'unità di misura utilizzata da *Kubernetes* per la misurazione dell'utilizzo della CPU è il millicore (1000m = 1 Core), nel nostro caso la soglia di utilizzazione è posta a 200m (1/5 di 1 Core) per il Server Web mentre per gli altri microservizi è di 100m (1/10 di 1 Core).

C. Minikube

Minikube è un tool per l'esecuzione di un cluster di *Kubernetes* in locale.

D. gRPC

gRPC è un moderno sistema di remote-procedure-call sviluppato da Google. All'interno della nostra applicazione è utilizzato per la comunicazione sincrona tra microservizi.

E. RabbitMQ

RabbitMQ è un middleware di comunicazione orientato ai messaggi che implementa il protocollo Advanced Message Queuing Protocol (*AMQP*). All'interno di tale applicazione questo middleware di comunicazione realizza una ricezione di messaggi di tipo selettiva mediante l'utilizzo di un *Exchange* diretto (Direct).

Le code create all'interno di *RabbitMQ* sono tre:

- *request_queue*: una per la richiesta della lista delle email dei partecipanti ad uno specifico gruppo, da *Shared Password* a *Group Manager*.
- *info_queue*: una per lo scambio delle informazioni riguardanti i gruppi, da *Group Manager* a *Shared Password*
- *email_queue*: una per le informazioni da includere nelle email, da *Shared Password* a *Notification*.

Tali code e l'*Exchange* sono stati creati con il flag che ne permette la persistenza. Inoltre, per ottenere la persistenza dei messaggi, è stata impostata una proprietà per il protocollo *AMQP* (Delivery Mode 2). Così facendo in caso di caduta del server *RabbitMQ*, al rilancio i messaggi vengono recuperati.

F. Helm Chart

È un package manager che semplifica le attività di deploy all'interno del cluster di *Kubernetes*.

Consente in particolare di:

- Recuperare pacchetti software da repository
- Configurare il deploy del software
- Installare e aggiornare software e le relative dipendenza

Nell'applicazione è stato utilizzato per l'installazione del package *MySQL* di *Bitnami*. Tale package fornisce dei file *yaml* configurabili per la replicazione del database relazionale *MySQL*. La replicazione in questione viene effettuata creando una replica primaria su cui vengono effettuate tutte le scritture e diverse repliche secondarie per le letture.

VII. Limitazioni riscontrate

A. Problema replicazione MySQL primary

Per la replicazione dei database *MySQL* è stato riscontrato il seguente problema: non è possibile avere più di una replica primaria su cui possano essere effettuate le scritture. Questo comporta che tutte le ulteriori repliche presenti nel cluster saranno sole repliche di lettura, le quali aggiorneranno il proprio contenuto in base alla replica primaria. La consistenza delle informazioni è stata gestita all'interno dei file *yaml* offerti da *MySQL* di *Bitnami*. All'interno di questi è possibile configurare il numero di repliche e l'utilizzo o meno di *Persistent Volumes*.

VIII. LIBRERIE

A. *Circuit-breaker - pybreaker*

È un'implementazione in *Python* del design patter *Circuit Breaker*. Tale libreria, è stata utilizzata prima di ogni invocazione alle chiamate *gRPC*.

B. *RabbitMQ - pika*

È un'implementazione in *Python* per il protocollo *AMQP* utilizzato da *RabbitMQ*. Tale libreria, è stata utilizzata per l'intero sistema di comunicazione asincrona nell'applicazione, tra cui le azioni di *publish* e *consume*.

C. *gRPC - grpcio*

È uno strumento *Python* che include il compilatore di *protocol buffer* ed un plug-in per la generazione dei codici server e client a partire dai file .proto. Tale libreria, è stata utilizzata per l'intero sistema di comunicazione sincrona nell'applicazione.

D. *Flask - flask*

È un framework *Python* per lo sviluppo di applicazioni web. Tale libreria, è stata utilizzata in entrambi gli *API gateway* per interfacciare l'utente all'applicazione.

E. *RestAPI - requests*

Mediante tale libreria, è stato possibile inviare richieste *HTTP/1.1* ai server web *Flask* ed interagire con *API REST*.